

ASSEMBLER/EDITOR OPERATION

Memory Space Requirements

The Assembler/Editor resides in memory in locations 2000 - 3800. In addition, the programs use locations 0300 - 0308 and 1E00 - 1FFF for workspace. Memory space must be allocated by the user for storage of the symbol table.

Reserving Space For The Symbol Table

The assembler will generate object code and store it at the locations specified in the assembly language program. RAM must exist in your APPLE at these locations. Since your source program is also memory-resident, your object program should not be written to the same RAM area wherein the source code resides.

In addition to space for source code and object code, you must reserve space in memory for the assembler to store its symbol table while it is constructing your object program. Each symbol you define in your source program will take 8 bytes of memory in the symbol table. For example, if you expect to use about 50 symbols in your source program, you should reserve at least 400 bytes for the symbol table. If the assembler runs out of room in the symbol table, the program will terminate with an error message.

You define the RAM area you wish to reserve for the symbol table by entering the upper and lower address limits of the area in two pairs of locations. At location 1FDF and 1FE0, you enter the low order and high order bytes, respectively, of the starting address of your symbol table. At locations 1FE1 and 1FE2, enter the location of the ending address of your symbol table. For example, to reserve locations 3802 through 3B00 for your symbol table, enter the monitor and type (when the APPLE monitor's asterisk prompt appears)

```
1FDF: 02 38 00 3B
```

If you specify these locations before entering the editor to type in your source program, you will be able to go directly to the assembler (with the A command) with the symbol table already defined.

If you elect to specify addresses for the symbol table after entering your source program, you must define the symbol table addresses before using the A command to call the assembler.

Reserved Memory Locations

The ARESCO Assembler/Text Editor uses all page zero memory locations. Object code should not be assembled into those locations. You may use page zero locations for data storage, but remember that the contents of those locations will be altered by the assembler and editor during their operations.

Assembling Large Source Programs From Disk Or Cassette Tape

If you wish to assemble a source program which will not fit all together in your available memory, you may enter it in segments, save the source code on disk or tape, and even assemble it in segments. (provided, of course, that the fully assembled object code will fit in your APPLE's available memory.)

Enter the first portion of your source program into the text editor and store it on audio cassette or disk (see the section on Saving Text On Audio Cassette Tape for details). Then enter the next and succeeding portions in the same source code memory area, designating each segment as a new file to the editor. Save each portion on audio tape. Only the last segment should contain an .END directive.

Now play the first segment back into the same source code memory area, using normal audio tape or disk loading procedures. Enter the editor, giving the base address of the text file and declaring it to be an old file. Assemble the first segment,

using the A command. (Remember to define your symbol table area first.) The assembler will return to the monitor when it comes to the end of the file. There is no need to save this assembled segment on tape or disk.

Now load and enter the second segment of your source code file into the same source code memory area, using the audio tape or disk loading procedure. Enter the editor, give the same base address as you used for the first segment, and declare it to be an old file. Do not use the A command to call the assembler. Exit to the APPLE monitor, and enter the assembler via location 3782 (type 3782G). The assembler will continue to assemble your program - right where it left off with the previous file - without clearing the previously generated symbol table or destroying the previously assembled program segment.

Continue to enter successive segments of text, entering the editor to give the base address each time. Exit to the monitor and re-enter the assembler at 3782. When the assembler encounters the .END directive, it will print the symbol table and terminate the assembly process. This is the time to save the assembled object code on APPLE disk or cassette.

INSTRUCTION FORMAT

Assembler instructions for the ARESCO Assembler are of two basic types, according to function:

1. Machine Instructions
2. Assembler Directives

Machine instructions correspond to the 55 operations implemented on the MOS 6502. The instruction format is

(label) opcode (operands) (comments)

Fields are in parentheses to show that they are optional. Labels and comments are always optional and many operation codes (opcodes) such as RTS (Return from subroutine) do not require operands. A typical instruction using all four fields is:

LOOP LDA BETA,X FETCH BETA INDEXED BY X

A field is defined as a string of characters separated by a blank space or tab character or characters. The list of opcodes for the ARESCO Assembler is shown in Table 1.

A label is an alphanumeric string of from one to six characters, the first of which must be alphabetic. A label may not be any of the 55 opcodes and may not be any of the special single characters A, S, P, X, or Y. These special characters are used by the assembler to reference the Accumulator (A), the Stack pointer (S), the processor status (P), and index registers X and Y respectively. A label may begin in any column, provided it is the first field of an instruction. Labels are used on instructions such as branch targets and on data elements for reference in operands.

The operands portion of an instruction specifies either an address or a value. An address may be computed by expression evaluation, and the assembler allows considerable flexibility in expression formation. An assembly language expression consists of a string of names and constants separated by operators +, -, *, and / (add, subtract, multiply, and divide). Expressions are evaluated left to right, with no operator precedence and no parenthetical grouping. Expressions are evaluated at assembly time and not at execution time.

Any string of characters following the operands field is considered to be comments, and is listed but not processed further. If the first non-blank character on a line is a semi-colon (;), the line is processed as a comment. On instructions which require no operand, comments may follow the opcode. A semi-colon need not precede the comment if the comment is on the same line as an instruction. At least one space must separate the fields of an instruction.

There are five assembler directives used to reserve storage and direct information to the assembler. Four have symbolic names with a period as the first character. The fifth, a symbolic equate, uses an equals sign (=) to establish a value for a symbol. A list of the directives is given here, but their use is explained later in the section on Assembler Directives.

```
.BYTE
.WORD
.OPT
.END
=
```

Labels and symbols other than directives may not begin with a period.

When using the assembler, remember that if you press any key during the assembly, the assembly and listing will stop, and you will be returned to the editor.

A typical assembler program segment is shown on the following page to illustrate the form of the information provided by the assembler. The formatting of text is done automatically unless you specifically select the NOTAB option (see Assembler Directives).

Note the semi-colons preceding comments which occupy separate lines (that is, do not occupy lines containing opcodes).

Constants

Constant values in assembly language can take several forms, as needed by the programmer. If a constant is other than decimal, a prefix character is used to specify type.

\$ (Dollar sign)	specifies hexadecimal
@ (Commercial "at")	specifies octal
% (Percent sign)	specifies binary
' (Apostrophe)	specifies an ASCII literal in immediate mode instructions
# (Pounds sign)	Specifies immediate mode

The absence of a prefix symbol indicates decimal value. In the statement

```
LDA BETA+5
```

the decimal number 5 is added to BETA to compute the address. Similarly, in the statement

```
LDA BETA+$5F
```

the hexadecimal value 5F is to be added to BETA for address computation. If you wish to load an ASCII character (for example, the letter "G") into the accumulator, you would use the apostrophe (single quote), as in the statement

```
LDA #'G
```

It isn't necessary to use a closing quote unless you have embedded quotes in a string of characters. To embed quotes, type the apostrophe twice (') rather than use the double quotes ("), at both the beginning and the end of the string.

Note that constant values can be used in address expressions and as values in immediate mode addressing. They can also be used to initialize locations, as explained later in a section on assembler directives.

Symbols

Symbols may be used to refer to addresses or to data values. Symbols must begin with an alphabetic character, and must be no more than six characters in length. The letters A, S, P, X, and Y may not be used, as previously explained.

ADDRESSING MODES

Symbolic

Perhaps the most common operand addressing mode is the symbolic form, as in

```
LDA BETA    PUT BETA VALUE IN THE ACCUMULATOR
```

In the example, BETA refers to a byte in memory that is to be loaded into the accumulator. BETA is an address at which a value is located. Similarly, in the instruction

```
LDA ALPHA+BETA
```

the address ALPHA+BETA is computed by the assembler, and the value at the computed address is loaded into the accumulator. Both ALPHA and BETA must have been previously defined.

Memory associated with the 6502 processor is segmented into pages of 256 bytes each. The first page, page zero, is treated differently by the assembler and by the processor, for optimization of memory storage space. Many of the 6502 instructions have alternate operation codes if the operand address is in page zero memory. In such cases, the address requires only one byte of storage rather than the normal two bytes. For example, if BETA is located at byte 4B in page zero memory, the code generated for the instruction

```
LDA BETA
```

is A5 4B

This is called "page zero addressing". If BETA is at 01 3C in page one memory, the code generated is

```
AD 3C 10
```

This is an example of "absolute addressing". Thus, to optimize storage space and execution time, a programmer should design with data areas in page zero memory whenever possible. Note that the assembler makes decisions on which form of the operation code to use based upon operand address computation.

Absolute Mode

In absolute mode, a specific address is given from which data is to be fetched or to which a branch will be made. Any two-byte address may be used, in decimal, hexadecimal, binary, or octal, provided the appropriate prefix character is employed.

Immediate Mode

It is often useful to be able to reference the address of a label as immediate mode data. The assembler recognizes the characters < and > for this purpose. For instance,

```
LDA    #< HERE
```

will load the accumulator with the low order eight bits of the address of the byte labeled HERE, and

```
LDA    #> HERE
```

will load the accumulator with the high order byte of the address of HERE.

Immediate mode addressing always generates two bytes of machine code, the opcode and the value to be used as operand. Note that constant values can be used in address expressions and as values in immediate mode addressing.

Relative Mode

There are 8 conditional branch instructions available to the user. An example is

```
BEQ      START      IF EQUAL BRANCH TO START
```

which might typically follow a compare instruction. If the values compared are equal, a transfer to the instruction labelled START is made. The branch address is a one byte positive or negative offset which is added to the program counter during execution. At the time the addition is made the program counter is pointing to the next instruction beyond the branch instruction. A branch address must be within 129 bytes forward or 125 bytes backward from the conditional branch instruction. An error will be flagged at assembly time if a branch target falls outside the bounds for relative addressing. Relative addressing is used only for branch instructions.

Implied Mode

Twenty-five instructions such as TAX (Transfer contents of Accumulator to Index X) require no operand and hence are single byte instructions. Thus, the operand addresses are implied by the operation code.

Four instructions (ASL, LSR, ROR, and ROL) are special, in that the accumulator, A, can be used as an operand. In this special case these four instructions are treated as implied mode addressing and only an operation code is generated.

Indexed Mode

Operands may be indexed with values in registers X and Y. Indexing is indicated by a comma and appropriate letter following the operand. For example

LDA BETA,Y

The value in register Y is added to BETA to form the address of the operand. Not all instructions can be indexed and on some,

indexing may be permitted with one register, but not the other. Refer to Table 2 for allowable addressing modes.

Indexed Indirect Mode

In this mode the operand address is a location in page zero memory which contains the address to be used as an operand.

An example is:

LDA (BETA,X)

The parentheses around the operand indicate it is indirect mode. In the above example the value in index register X is added to BETA. That sum must reference a location in page zero memory. During execution the high order byte of the address is ignored, thus forcing a page zero address. The two bytes starting at that location in page zero memory are taken as the address of the operand. For purposes of illustration, assume the following:

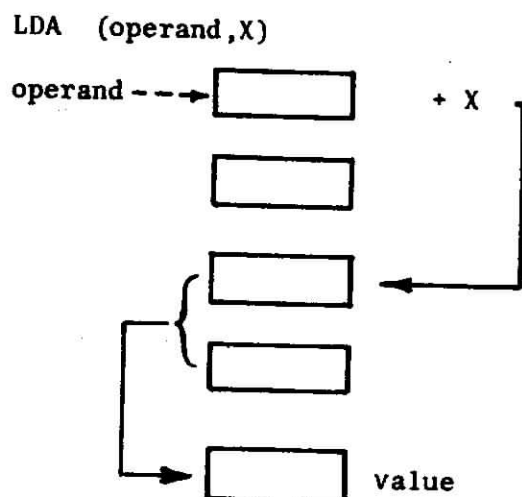
BETA is 12

X contains 4

Locations 0017 and 0016 are 01 and 25

Location 0125 contains 37

Then BETA + X is 16, the address at location 16 is 0125. The value at 0125 is 37 and hence the instruction LDA (BETA,X) loads the value 37 into the accumulator. This form of addressing is shown in the illustration below.



Indirect Indexed Mode

Another mode of indirect addressing uses index register Y and is illustrated by:

LDA (GAMMA),Y

In this case GAMMA references a page zero location at which an address is to be found. The value in index Y is added to that address to compute the actual address of the operand. Suppose for example that:

GAMMA is 38 (hexadecimal)

Y contains 7

Locations 0039 and 0038 are 00 and 54

Location 005B contains 126

Then the address at 38 is 0054 and 7 is added to this, giving an effective address 005B. The value at 005B is 126 which is loaded into the accumulator.

In indexed indirect the index X is added to the operand prior to the indirection. In indirect indexed the indirection is done and then the index Y is added to compute the effective address. Indirect mode is always indexed except for a JMP instruction which allows an absolute indirect address as exemplified by JMP (DELTA) which causes a branch to the address at location DELTA. The indexed indirect mode of addressing is shown in the illustration below.

LDA (operand),Y

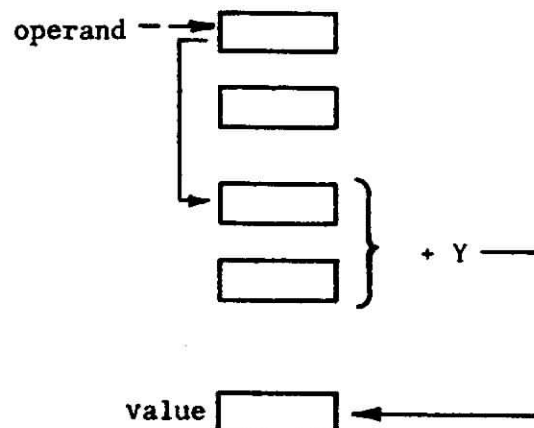


Table 2
Instruction Addressing Modes

	Immediate	Page zero	Absolute	Page zero indexed by X	Absolute indexed by X	Absolute indexed by Y	Indirect indexed
ACD	X	X	X	X	X	X	X
AND	X	X	X	X	X	X	X
ASL (1)		X	X	X	X		
BIT		X	X				
CMP	X	X	X	X	X	X	X
CPY	X	X	X				
CPX (2)	X	X	X				
DEC		X	X	X	X		
EOR	X	X	X	X	X	X	X
INC		X	X	X	X		
JMP (3)			X				X
JSR			X				
LDA	X	X	X	X	X	X	X
LDX (2)	X	X	X	X	X		
LDY	X	X	X	X	X		
LSR (1)		X	X	X	X		
ORA	X	X	X	X	X	X	X
ROL (1)		X	X	X	X		
ROR (1)		X	X	X	X		
SBC	X	X	X	X	X	X	X
STA		X	X	X	X	X	X
STX		X	X	X			
STY		X	X	X			

(1) Accumulator A can also be an operand

(2) Indexing with Y

(3) Indirect is absolute indirect and not indexed

ASSEMBLER DIRECTIVES

There are five directives which are used to control the assembly process, define values or initialize memory locations. Assembler directives always appear in the opcode field of an instruction and thus might be considered as assembly time opcodes instead of execution time opcodes. The directives are: .BYTE, .WORD, .OPT, .END and equates (which is denoted by the equals sign =). All directives which are preceded by the period may be abbreviated to the period and three characters if desired (eg., .BYT).

.BYTE is used to reserve one byte of memory and load it with a value. The directive may contain multiple operands which will store values in consecutive bytes. ASCII strings may also be generated by enclosing the string with quotes.

```
HERE      .BYTE  2
THERE     .BYTE  1, $F, @3, %101, 7
ASCII     .BYTE  'ABCDEFH'
```

Note that numbers may be represented in the most convenient form. In general, any valid MCS650X expression which can be resolved to eight bits may be used in this directive. If it is desired to include a quote in an ASCII string, this may be done by putting two quotes in the string;

```
.BYTE 'JIM''S CYCLE'
```

could be used to print:

```
JIM'S CYCLE
```

.WORD is used to reserve and load two bytes of data at a time. Any valid expression, except for ASCII strings, may be used in the operand field.

```
HERE     .WORD  2
THERE    .WORD  1, $FF03, @3
WHERE    .WORD  HERE, THERE
```

The most common use for .WORD is to generate addresses, as shown in the above example. "WHERE" stores the 16 bit addresses of "HERE" and "THERE". Addresses in the 6502 are fetched from memory in the order low-byte, high-byte, and therefore .WORD generates the values in this order. The hexadecimal portion of the second example above (\$FF03) would be stored as 03 FF.

= is the EQUATE directive, and is used to reserve memory locations, reset the program counter (*), or assign a value to a symbol.

HERE **+=1	reserve one byte
WHERE **+=2	reserve two bytes
**=\$200	set program counter
NB=8	assign value
MN=Nb+%101	assign value

Note that expressions must not contain forward references or they will be flagged as an error. For example,

Q=C+D-E*F

is legal only if C, D, E, and F are all defined. It is illegal if any of the variables is a forward reference, to be defined later in the program. Forward references in expressions may be used in the modified two-pass version of the ARESCO Assembler, but not in the one-pass version.

Note also that expressions are evaluated in strict left to right order.

.OPT is the most powerful directive available, and is used to control generation of output fields, listings, and expansion of ASCII strings in .BYTE directives. The options available are:

.OPT	ERRORS, LIST, SYMBOLS, GENERATE, TAB
.OPT	NOERRORS, NOSYMBOLS, NOLIST, NOGENERATE, NOTAB